第2章　データベースとWEBサービス構築

担当者： 坂本大介（ヒューマンコンピュータインタラクション研究室）
部屋： 情報エレクトロニクス棟 8-13　内線：7225
E-mail: sakamoto@ist.hokudai.ac.jp

1　はじめに

Web プログラミングは Web サイトに特化したプログラミングのことです．対象となるサービスの例として，世界中で利用され，皆さんにも馴染みのあるショッピングサイトやブログサービス，ソーシャルネットワークゲームなども含まれ，小規模なものから大規模なものまで様々です．これらの Web サービスは，Web ブラウザを通したユーザの要求に対し，サーバ側でデータベースから情報を取得し，それを加工して，Web ブラウザを通してユーザに返すという仕組みになっています．この流れを大きく 2 つに分けると，サーバ側（サーバサイド，バックエンド）とクライアント側（フロントエンド）に分かれます．また，Web サービスの構築には，データベースの扱いや Web アプリケーションを構築するスクリプト言語など，大量のアクセスや処理に備えた負荷分散の知識，各種セキュリティ，WEB サーバの設定など，必要な知識は多岐に渡ります．このように，Web プログラマはその他のプログラマ職に較べ，習得が必要な言語が多くありますが，比較的習得が容易なものが多いため，独学で習得することが可能です．今回の実験を通して，自分自身で Web プログラミングスキルを習得できることや，能力を高めることができることを理解して下さい．

1.1　実験の流れ (6 日間)

1 日目： フレームワークなしの Web サーバプログラミング①

2 日目： フレームワークなしの Web サーバプログラミング②　（課題有り）

3 日目： データベース

4 日目： データベース/フレームワークを使った Web プログラミング （課題有り）

5 日目： フレームワークを使った Web プログラミング (Sinatra)

6 日目： フレームワークを使った Web プログラミング (Sinatra) （課題有り）

1.2　レポート

各章について，全ての基本課題と少なくとも一つの応用課題を解くこと. 各章毎に取り組んだ課題についてのレポートを作成すること．レポートの最初には必ず学部番号，氏名，提出日を記載すること．レポートには，課題に対する結果・説明などを記入すること．提出期限についてはそれぞれ指示する．

レポートやソースコードなど全てのファイルを zip ファイルにまとめて提出すること．zip ファイルの作成方法は Moodle で指示する．

【レポート提出方法】ELMS 上の本講義に関する Moodle で各章毎の提出フォームを用意しておくので，そこに zip ファイルをアップロードすること．

1.3 実験を行う際の注意事項

ディレクトリやファイルを命名する際は必ず英語で作成してください．文字コードの違いにより，エラーが起きる可能性があります．(パスに日本語が含まれていてもエラーとなる可能性があります！)

2 フレームワークなしの Web サーバプログラミング（1 日目,2 日目）

2.1 Ruby（再？）入門

Ruby 概要

Ruby はプログラミング言語の一つで，Perl や Python，PHP などと同様なインタプリタ型の言語です．Ruby は，豊富なライブラリ，日本語のドキュメント（`https://docs.ruby-lang.org/ja/`）などの特徴があり，Web プログラミングでもよく使われる言語ですので，この実験を機会に習得しましょう．

実行方法

前述のように，Ruby はインタプリタ型言語です．Ruby のプログラムを実行するには様々な方法がありますが，ここでは二つの方法を紹介します．

まずは対話型の irb (Interactive Ruby) コマンドを使う方法です．Linux のコマンドプロンプトで，下のコマンドを入力してみてください（以降，$ はコマンドプロンプトを表すとします）．

```
$ irb
irb(main):001:>
```

これが irb です．では，プログラムを実行してみましょう（irb(main) 等の表示がない行は出力結果です.）．

```
irb(main):001:0>1+1
  2
irb(main):002:0> 6*7
  42
irb(main):003:0> 6-7
  -1
irb(main):004:0> 22/2
  11
```

irb を終わるには，exit と入力します．

```
irb(main):005:0>exit
```

もう一つの実行方法は，ファイルを経由する方法です．まず，下のソースコードを **test.rb** という名前で，ファイルに保存してください（emacs なり vim なり gedit なり，好きなエディタで）．

test.rb

```
# comment line
puts("1+1 = ")
puts(1 + 1)
```

「#」で始まる行はコメントを表します．次のコマンドで実行しましょう．

```
$ ruby test.rb
  1+1 =
  2
```

以降のプログラム例では，特に指定している場合（ソースコード先頭にファイル名があるなど）を除き，irb とファイルの好きな方で実行してください．

2.2 今回使う Ruby

変数

C 言語と同じように変数があります．ただし，C 言語とは異なり，

- 変数宣言は必要ない
- 変数に型がない (動的型付け)

といった性質があります（Python や Perl と同じです）．

```
var1 = 10
puts(var1)
```

前述のように変数に型が無いので，整数 (Integer) を代入した後，浮動小数点型 (Float) を代入することも出来ます．

```
var1 = 10
puts(var1)
var1 = 1.5
puts(var1)
```

メソッド

C 言語での関数に相当するものを，Ruby ではメソッド (Method) と呼びます．先ほどから使っている “puts” も，実はメソッドです．

メソッドでは,「レシーバ」と呼ばれるオブジェクトを「.」の左側に置きます．

```
puts(10.even?) # true
puts(11.even?) # false
```

上の例では「10」,「11」がレシーバで,「even?」がメソッドです（puts ではレシーバが省略されています.）．

メソッドは，レシーバと引数の両方を取ることも出来ます．

```
puts(42.gcd(12)) # gcd (n) は 自身と整数 n の最大公約数を返します。
```

なお，Ruby では多くの場合，引数前後の括弧は省略出来ます．

```
puts("hoge")
puts "hoge"
```

メソッドと関数，何が違うかが気になる人は，オブジェクト指向プログラミングについて勉強してください．

条件文とループ

もちろん，条件文とループもあります．まずは，条件文 if の例です．

```
var = 10
if var == 10
  puts("I'm number ten")
elsif var == 9
  puts("I'm number nine")
else
 puts("I'm not ten")
end
```

else や elsif が無い if だけの場合，一行で書くことも出来ます．

```
var = 9
puts "I'm number nine" if var == 9
```

次にループしましょう．一番基本的なループ方法は，while 文です．

```
i = 0
while (i< 10)
  puts i
  i = i + 1
end
```

他にも，様々な方法があります．

```
10.times do |i|
  puts i
end

0.upto(10) { |i|
 puts i
}
```

「do〜end」「{ }」の部分は「ブロック」と呼ばれ，どちらも（ほとんど）同じように使えます．「 |i|」の部分は，| | の間の変数（ここでは i）に値を代入することを表します．ここで使われている変数は，「do」と「end」の外では使えない点は注意点です（詳しくは「ruby 変数のスコープ」などで検索してください）．

文字列

ほとんどの言語と同様，Ruby にも文字列があります

```
puts "hoge"
puts 'foo'
```

一行目ではダブルクオート「"」，二行目ではシングルクォート「'」を使っていることに注意してください．

シングルクオート「'」と違い，ダブルクオート「"」では変数の埋め込み出来ます．

```
answer = 42
puts "The answer to the ultimate question is #{answer}"
puts 'The answer to the ultimate question is #{answer}'
```

上の例では,「42」という数値が文字列に変換されています．明示的に文字列に変換するには「to_s」，逆に整数に変換するには「to_i」というメソッドを使います．

```
var1 = "6"
var2 = "7"
puts var1.to_i + var2.to_i
puts var1 + var2
```

Ruby では，文字列に対する「+」は，文字列の連結を表します．

入出力

これまでのプログラミング例は，データが固定されていました．キーボードから入力を取得してみましょう．

```
var = gets().chomp #「.chomp」で改行文字を取り除く
if var == "hoge"
 puts "buzz"
end
```

ファイル出力は以下のように行います．

```
f = File.open("hoge.txt", "w")
f.puts("hoge buzz foo")
f.close
```

カレントディレクトリの hoge.txt が作成されているはずです．ファイルの入力は以下のように行います（上の hoge.txt が作られているディレクトリで実行してください）．

```
f = File.open("hoge.txt", "r")
f.each_line do |line|
 puts line
end
f.close
```

配列，ハッシュ

ほとんどのプログラミング言語と同様，配列もあります．コード例を下に示しますので，実行して確認してください．

```
arr = [1,2,3] # 初期化
puts arr[0]   # 最初の要素
puts arr[-1]  # 最後の要素
puts arr      # arr の出力
arr << 1192   # 「1192」を追加
puts arr      # arr の出力
arr.each do |x|  # イテレーション
 puts x
end
```

最後の例は，ループを抽象化したイテレーションと呼ばれる機能です．

配列と似た，ハッシュ(Hash) というデータ構造もあります．配列と違い,「任意」のデータを添え字に使うことが出来ます．

```
h = {"a" => 1, 2 => "b", 3.14 => "pi"}
puts h["a"]
puts h[2]
# イテレーション
h.each do |key, value|
 puts "#{key} => #{value}"
end
```

Ruby のハッシュでは，コロン「:」を前に置いた記号列，シンボル（Symbol）がキーとして好んで使われます．

```
h = {:a => 1, :b => 2, :c => 3}
puts h[:a]
puts h[:b]
```

ライブラリ

Ruby には標準で色々なライブラリが組み込まれています（一覧はリファレンスマニュアル `https://docs.ruby-lang.org/ja/latest/library/index.html`）．ライブラリを呼び出すには，require もしくは load 文を使います．

試しに，素数を扱う prime ライブラリを使ってみましょう．

```
require "prime"
puts "12 = #{12.prime?}"
puts "13 = #{13.prime?}"
pg = Prime::EratosthenesGenerator.new
30.times do |i|
 puts pg.next
end
```

ちょっと素数が欲しい時に便利ですね．なお，Prime::EratosthenesGenerator.new の「::」の部分は，とりあえず「名前が被らないようにしている」ぐらいに思っておいてください

2.3 サーバ・クライアント通信を作ってみる

Web アプリケーションを作成する前段階として，基礎的なサーバ・クライアント通信を行うプログラムを作成します．

Web 通信復習

授業で習ったと思いますが Web 通信について簡単に復習しましょう．トランスポート層として TCP，アプリケーション層で HTTP を使って通信します．TCP はサーバとクライアントが「ハンドシェイク」で接続を確立したのち，通信を行います．もう一つのメジャーなプロトコルである UDP と比較して，パケットの整理を行う点が TCP の特徴です．HTTP では,「GET」と「POST」というメソッドを主にリクエストします.「GET」はサーバの状態やデータを（あまり）変えないリクエスト,「POST」はサーバの状態やデータを変えるリクエストに使います．Web ブラウザはサーバにリクエストを送信し，サーバは処理した結果を，HTML などでクライアントに返します．ブラウザは受け取ったデータを解析し，適当な処理（HTML なら DOM の解析，修飾など）を行い，ユーザに表示します．

TCP サーバを作る

TCP サーバとは，TCP/IP 通信により，クライアントと通信を行うサーバです．まずは以下のコードを入力してみましょう．

TCP_Server.rb

```
# coding: utf-8
require 'socket'

server = TCPServer.new 2000  #2000 番ポートにサーバを立てる

#クライアントの受付開始
client = server.accept
#サーバのプログラムは，クライアントからのリクエストがあるまでここで一時停止（ブロッキングします）

client.puts 'Hello, client' #クライアントへ文字列を送信
puts client.gets #クライアントから受信した文字列を表示
client.close #クライアントとの通信を切断
```

TCP_Client.rb

```
# coding: utf-8
require 'socket'

socket = TCPSocket.new('127.0.0.1', 2000) #127.0.0.1 の 2000 番ポートのサーバに接続
puts socket.gets #サーバから受信した文字列を表示
socket.puts 'Hello, server' #サーバへ文字列を送信
socket.close #サーバとの通信を切断
```

プログラムを実行するには，

1. 一つのコマンドプロンプトで「$ruby TCP_ Server.rb」
2. 一つのコマンドプロンプトで「$ruby TCP_ Client.rb」

とタイプします．起動する順番に気をつけてください．

IP アドレス 127.0.0.1（ローカルループバックアドレス）の 2000 ポートでサーバを開いています．ここの 2000 という番号に特に意味はなく，サーバとクライアントが同じ番号で通信をしているという事が重要です．なお，特定の用途に使われるポート（Well-known port）もありますので，注意してください．

HTTP サーバを作る

次は，HTTP を用いて，ブラウザで開くことができる HTTP サーバを作成しましょう．私達が普段見ている Web ページは，HTTP を用いて表示されます．HTTP には様々なリクエストが存在し，クライアント（Web ブラウザなど）がそれらを使い分けることで，目的の情報をサーバから受け取ることができます．この節では，一番簡単な GET リクエストに対して，HTML 文書を返すプログラムを作成します．以下が HTTP サーバのプログラムです．

HTTP_ Server.rb

```
# coding: utf-8
require 'socket'

server = TCPServer.new 2000
loop do
  client = server.accept
  headers = []
  while header = client.gets
    break if header.chomp.empty?
    headers << header.chomp
  end
  p headers

  client.puts "HTTP/1.0 200 OK"
  client.puts "Content-Type: text/html"
  client.puts
  client.puts "<h1>Hello, World!</h1><h2>Welcome!</h2><p>This is my personal
page!</p>"
  client.close
end
```

ブラウザで確認

HTTP サーバのプログラムを作成したら，実際にサーバを起動してブラウザで確認してみましょう．サーバを起動したら，ブラウザ（Firefox 推奨）を開き，URL の欄に「127.0.0.1:2000」と入力しましょう．Web ページの画面が出てきたら成功です！また，サーバ側の端末を見てみると，ブラウザから与えられたリクエストが表示されています．

クライアントで確認

次は，クライアント側のプログラムから，HTTP サーバへリクエストを送ってみましょう．HTTP サーバを起動した状態で，以下のプログラムを実行してみましょう．

HTTP_Client.rb

```
# coding: utf-8
require 'socket'

socket = TCPSocket.new('127.0.0.1', 2000)
socket.write "GET / HTTP/1.0\r\nAccept: */*\r\nConnection: close\r\nHost:
127.0.0.1:2000\r\n\r\n"

response = '' # これはシングルクォーテーション「'」が二つで，初期化です
while t = socket.read(1024)
  response = response + t
end

puts response
```

クライアント側のプログラムを実行することで，得られたプログラムは，HTML 文書がそのまま表示されていることが確認できます．Web ブラウザは，これを自動的に変換することで，普段私達が見ているような見やすい Web ページを表示することができるのです！

2.4 Webrick で Web サーバを作ってみる

Webrick は Ruby でサーバを作るためのライブラリです．Socket を使うよりも楽に Web サーバを作ることが出来ます．ちなみに，この実験の後半で学ぶ Sinatra などの Web フレームワークも，Webrick を（デフォルトでは）使って実装されています．

Webrick ことはじめ

まずは，ページを一つだけ処理するプログラムを書いてみましょう．

webrick_plain.rb

```
require 'webrick'
srv = WEBrick::HTTPServer.new({ :DocumentRoot => './',
  :BindAddress => '127.0.0.1',
  :Port => 2000})
srv.mount('/hoge', WEBrick::HTTPServlet::FileHandler, 'hoge.txt')
trap("INT"){ srv.shutdown }
srv.start
```

ブラウザで「127.0.0.1:2000/hoge」を開いてみてください．見れましたか？プログラムは Ctrl+C で終了出来ます．

処理内容を説明すると，

- IP アドレス 127.0.0.1（ローカルループバックアドレス）の 2000 で動くサーバを作る
- 127.0.0.1:2000/hoge というリクエストに対して，hoge.txt というファイルを返す設定
- INT シグナル（Ctrl+C で送られるシグナル）を受信すると，サーバをシャットダウンする

となります．

HTML

Web ブラウザで閲覧するページの多くは，HTML で構造化，CSS で装飾されています．残念ながら，詳しく説明するスペースがありませんが，よく用いられる HTML タグを表に列挙してみましたので，参考までに．

タグ	用途
a	リンク
h1∼h6	見出し
p	段落
div, span	領域
img	画像挿入
script	Javascript
table, td, tr	表関連

Webrick を使い，この HTML ファイルを送るようにしてみましょう．まず，下の HTML ファイルを test.html というファイル名で保存して，ブラウザで開いてみてください．

test.html

```
<!DOCTYPE html>
<html lang="ja">
    <head>
      <meta charset="UTF-8">
    </head>
    <body>
        <h1>テスト</h1>
        <p>これはテストです．いいえ，それはペンです．</p>
        <h2>後藤さんの話</h2>
        <p>これはペンです．いいえ，それは後藤さんです．</p>
        <h1>リンクの例</h1>
        <p><a href=”http://example.com/”>これ</a>は後藤さんではありません．サンプルページです．</p>
    </body>
</html>
```

次に以下のソースコードを実行して，Web サーバ経由で HTML ファイルを見られることを確認してみてください．

webrick_html.rb

```
require ’webrick’
srv = WEBrick::HTTPServer.new({ :DocumentRoot => ’./’,
  :BindAddress => ’127.0.0.1’,
  :Port => 2000})
srv.mount(’/test’, WEBrick::HTTPServlet::FileHandler, ’test.html’)
trap("INT"){ srv.shutdown }
srv.start
```

動的な処理

!!!!WARNING!!!!

以下のコードにはセキュリティ上の問題があります（課題参照）

これまでは，リクエストに対し固定したファイルを返す処理だけでしたが，実際の Web では様々な動的な処理も行われています．動的な処理には，クライアント側もサーバ側もありますが，ここでは CGI を用いたサーバ側の処理を紹介します．CGI(Common Gateway Interface) は，Web サーバと外部プログラムを連携させる仕組みで，渡されたパラメータに応じた動的な処理が可能となります．

まず，サーバ webrick_ cgi.rb を作成してください．

webrick_cgi.rb

```
require 'webrick'
CGI_PATH = '/usr/bin/ruby'
srv = WEBrick::HTTPServer.new({ :DocumentRoot => './',
  :BindAddress => '127.0.0.1',
  :Port => 2000,
  :CGIInterpreter => CGI_PATH})
srv.mount('/cgi', WEBrick::HTTPServlet::CGIHandler, 'cgi.rb')
trap("INT"){ srv.shutdown }
srv.start
```

（必要に応じ，ruby のパスを whereis コマンドで調べ，CGI_ PATH の部分を変更してください.）

次に cgi.rb というファイル名で，サーバのファイルと同じディレクトリに保存してください．

cgi.rb

```
require "cgi"
cgi = CGI.new

cgi.out("type" => "text/html" ,
      "charset" => "UTF-8")  do
 # パラメータ取得
  msg = cgi["message"]
  iteration = cgi["iter"].to_i
  html = "<html><body>\n"
  iteration.times do |i|
  html = html + "<p>#{i}: #{msg}</p>"
  end
  html = html + "</body></html>"
  html
end
```

このプログラムを実行するには，以下のコマンドで起動し (cgi.rb サーバから適宜呼ばれます)，「127.0.0.1:2000/cgi?message=hoge&iter=3」を Web ブラウザで開いてみてください．

```
> ruby webrick_cgi.rb
```

「message」の部分が 3 回表示されましたか？

「?」以降の「message=hoge&iter=3」の部分が URL パラメータで,「パラメータ名=値」が「&」で連結されています．CGI スクリプトで，このパラメータを取得し，その内容に応じた処理を行っています．

クライアント側での動的な処理は，JavaScript という言語で実装出来ますが，紙面の都合で省略します．興味のある人は調べてみてください．

2.5 課題

下の基本課題全てと，応用課題を少なくとも一つ解きなさい．提出期限および提出先は Moodle 上で案内する．各課題には，ソースコード（必要にして十分なコメントを付けて），実行結果例，その他説明を記入しなさい．

基本課題 2-1

Ruby で Fizz Buzz のプログラム「**FizzBuzz.rb**」を書きなさい．FizzBuzz とは，与えられた数値が

- 3 で割り切れる時は,「Fizz」を
- 5 で割り切れる時は,「Buzz」を
- 15 で割り切れる時は,「Fizz Buzz」を
- それ以外の時は，その数を

表示する問題です．プログラムでは標準入力から整数を受け取り，1 からその数までの整数に対して FizzBuzz の結果を出力するようにしてください．

基本課題 2-2 Ruby ライブラリを試す

Prime，Socket，Webrick，CGI 以外の Ruby ライブラリを一つ調べ，それを使ったプログラム「**tryLibrary.rb**」を作成し，ライブラリの機能とプログラムの説明をしてください．
プログラムはコメント行と空行を除き，5 行以上，400 行以下で作成してください．

基本課題 2-3 Socket を使ったサーバ・クライアント

Socket 通信を使い（Webrick を使わずに），以下の条件を満たす TCP サーバ・クライアント「**minChat_Server.rb**」と「**minChat_Client.rb**」を作りなさい．

- サーバは 1 人のクライアントと Socket 通信を行う
- クライアント側から標準入力を行うと，その文字列がサーバ側に表示される
- クライアントが文字列「bye」を送信したら，通信を切断する

基本課題 2-4 Webrick 使ったサーバ

Webrick を使い，以下の条件を満たす Web サーバ「**reply_WebServer.rb**」を作りなさい．

- 「127.0.0.1:2000/time」というリクエストには，サーバの現在時刻を含む HTML を返す
- 「127.0.0.1:2000/fizzbuzz」というリクエストは,「num」パラメータを受け取り，1 から num までの数に対して fizzbuzz の結果を，HTML のテーブル形式で返す

基本課題 2-5 セキュリティ

動的な処理の例のソースコードには，セキュリティ的にあまり良くありません[1]．XSS(クロスサイトスクリプティング) と，CSRF（クロスサイトリクエストフォージェリ）について調べ

- 攻撃方法（XSS と CSRF 両方）
- 攻撃例（XSS のみ）
- 対策（XSS と CSRF 両方）

についてまとめなさい．また，XSS と CSRF 以外の Web プログラミングでのセキュリティ問題について一つ調べ，その攻撃方法と攻撃例，対策を述べなさい．

応用課題 2-1 掲示板

次の要件を満たす掲示板サーバ「**bulletinBoard_Server.rb**」を作りなさい．

1. サーバはクライアントから受け取ったメッセージを（好きな方法で）保存
2. クライアントはサーバに対し，書き込み「write」，書き込み一覧取得「index」，書き込み取得「read」のリクエストが出来る
3. 「write」は書き込むメッセージをパラメータとして受け取り，サーバはメッセージを保存し，ユニークな id を発行する
4. 「index」は保存されているメッセージの id 一覧を返す
5. 「read」は読み込む id をパラメータとして受け取り，その保存されているメッセージを返す

実行例を下に示します．

```
http::://127.0.0.1:2000/write?msg=nullpo
http::://127.0.0.1:2000/write?msg=ga
http::://127.0.0.1:2000/index
id:1
id:2
http::://127.0.0.1:2000/read?id=1
nullppo
```

また，上の要件以外にも，掲示板に必要な機能（禁止ワード，投稿削除，スレッド機能など）を少なくとも一つ実装しなさい．

応用課題 2-2 ログイン画面

次の要件を満たすログイン機能を持つ Web サーバ「**loginServer.rb**」を作りなさい．

1. 「login」というリクエストを受け付ける
2. 「login」リクエストでは「password」というパラメータを受け取り，パスワードを保存した List.txt の内容と比較する

[1]サンプルコードに対する XSS は実行出来ないと思います．その理由が気になる人は，X-XSS-Protection で調べてみてください．

3. List.txt にはパスワードのリストを，改行区切りで保存しておく（下記の例参考）

4. 「password」で渡されたパスワードが List.txt に保存されている場合と，保存されていない場合で，区別出来るようにレスポンスを返す

[ヒント] **List.txt** から文字の読み取りの処理

```
f = File.open("ファイル名", "r")
f.each_line do |line|
  #一行ずつ文字列を読み取り
end
f.close
```

一行ずつ読み取った情報の末尾に改行文字 “/n”,“/r” がついているので (文字列名).chomp で取り除きましょう.

List.txt ... 改行区切りで入力しておく

```
hoge
hirakegoma
```

「ho」や「hogehoge」はパスワードとして満たさない工夫が必要.

また，入力のフォームには以下の HTML を用い，Webrick もしくは Socket を用い，パラメータを受け取る login リクエストのプログラムを作成しなさい.

```
<!DOCTYPE html>
<html lang="ja">
  <head>
    <meta charset="UTF-8">
  </head>

  <body>
    <form action="login" method="POST" >
      <input type="text"  name="password" > <br>
      <input type="submit" >
    </form>
  </body>
</html>
```

3 データベース（3 日目,4 日目）

3.1 今回使う Ruby 構文

gem

Ruby には Gem と呼ばれるライブラリ群があり，RubyGems(https://rubygems.org/) では，サードパーティー製のライブラリを簡単に入手することが出来ます．Gem を使うことにより，ライブラリのインストール，アップデート，依存性管理などを簡単に行うことができます．特に RubyGems にある個々のライブラリを「パッケージ」と呼ぶことが一般的です．

まずは，現在 PC に入っているパッケージを確認してみましょう．以下のコマンドで確認することができます．

```
$ gem list --local
```

次は，実際にパッケージをインストールしてみましょう．本章で使用する sqlite3 をインストールします．以下のコマンドを入力してください．

```
$ gem install sqlite3
```

このように,「gem install パッケージ名」で任意のパッケージをインストールすることができます．

クラス

Ruby はオブジェクト指向言語ですので，クラスがあります．ひとつ，使ってみましょう．

testClass.rb

```
class A
 def initialize(msg)
  @msg = msg
 end
 def repeat_msg(iter)
  iter.times do |i|
   puts @msg
  end
 end
end
a = A.new("hoge")
a.repeat_msg(3)
```

少し説明すると，

- @msg の「@」はインスタンス変数
- 「def〜」はインスタンスメソッドの定義
- 「initialize」は特別なメソッドでコンストラクタの定義となり，new メソッドで呼び出すとなります．

その他

Ruby ではダブルクオーテーション「“ 」あるいはシングルクオテーション「‘ 」で囲んで文字列を表すと説明しましたが，この文字列中に特殊な文字を入れたくなる時があります．例えば,「“ 」をダブルクオーテーションで囲んだ文字列中で使うようなケースです．このような場合，バックスラッシュ「\」記法を用います．

```
puts "\"がダブルクオーテーションです"
```

次に，便利な join と split メソッドを紹介します．以下のプログラムを実行して，動作を確認してみましょう．

```
arr = [1,2,3]
str = arr.join(",")
puts str
arr2 = str.split(",")
arr2.each do |x|
 puts x
end
```

3.2 データベース

概要

何がしかの情報の集まりをデータベース (database) と呼びます．データベースを扱うソフトウェアには以下のような様々な機能が求められます．

- データの書き込み，検索
- データの一貫性の確保
- 障害への耐性，障害発生時の復旧
- 大量のデータへの対応

これらの機能を確保するための専門のデータベース管理システム (DataBase Management System、略称：DBMS) が，開発，利用されています．

RDBMS と SQL

DBMS には色々な種類がありますが（課題 3-1），この章で利用する関係データベース管理システム（英語: Relational DataBase Management System、略称：RDBMS）は現在主流の DBMS です．RDBMS では，関係モデルと呼ばれる考え方に基づき，データを表（テーブル）と，表の間の関係で管理します．

SQL は RDBMS を操作するためのデファクトスタンダードな言語です．言語と言っても考え方・書き方が Ruby や Java などの手続き型とは大きく異なり，問い合わせ型言語と呼ばれています．RDBMS+SQL はデータベースとしてよく用いられ，代表的なソフトとしては MySQL，PostgreSQL，Oracle，Microsoft SQL Server，そして今回紹介する SQLite などがあります．

3.3 SQlite

概要

今回の実験では，RDBMS の一つである SQLite を使用します．SQLite は，MySQL や PostgreSQL などと異なり，サーバ型では無い RDBMS です．軽量なデータベースとして，アプリケーションの組込み用途などにも使われています（例えば Firefox や Android）．

以下で紹介する内容の多くは，一部 SQLite 特有のものもありますが，ほとんどは他の RDBMS でも使えます．

インストールと操作方法

計算機室の PC にはインストールされているので，すぐに使うことが出来ます．なお，スタンドアロン型なので，個人の PC にも容易にインストール出来ると思います．

```
$ sqlite3
  SQLite version 3.7.17 2013-05-20 00:56:22
  Enter ".help" for instructions
  Enter SQL statements terminated with a ";"
sqlite>
```

終了は

```
sqlite>.exit
```

で行います．

SQlite の管理命令はピリオド「.」 で始まり，大文字小文字を区別し，セミコロン「;」が末尾に不要です．SQL 文とは反対なので注意しましょう．

「–」でそれ以降のコマンド,「/*」と*/」の間のコマンドは，コメントとなり，インタプリタから無視されます．

```
sqlite> -- no one sees the real me
sqlite>
sqlite> /* this is comment */
sqlite>
```

次に，練習に使うデータベースとテーブルを作りましょう．例に使うのは，成績表を表すテーブル (student_records) と，学生の情報を表すテーブル (students) です．

sandbox.db を作る

```
$sqlite3 sandbox.db
  SQLite version 3.7.17 2013-05-20 00:56:22
  Enter ".help" for instructions
  Enter SQL statements terminated with a ";"
sqlite3>CREATE TABLE student_records(id integer PRIMARY KEY, student_id integer,
klass text, score integer, grade text);
sqlite3>CREATE TABLE students(student_id integer PRIMARY KEY, name text);
```

カレントディレクトリに「sandbox.db」というファイルが作成されているはずです．このファイルがデータを保存しているファイルとなります．

「CREATE TABLE」はテーブル名，テーブルのカラム（列）の名前と型を指定し，データベースにテーブルを作成する命令で，この例では student_records と students のテーブルを作成しています．「PRMARY KEY」はテーブルの主キーを指定しています（主キーについては後述）．

SQL の命令文は大文字と小文字を区別しないのですが，慣例としてキーワードは大文字，カラム（列）名などは小文字を使うことが多いです．

以下に，作成したテーブルを示します．

student_records テーブル				
id(integer PRIMARY KEY)	student_id(integer)	klass(text)	score(integer)	grade(text)

ここで本来 “class” となるところが “klass” となっていますが，これは間違いではありません．詳細は 3.5 節を参照してください．

students テーブル	
student_id(integer PRIMARY KEY)	name(text)

CRUD: Create，Read，Update，Delete

データベースの操作で最も基本的な，Create，Read，Update，Delete をまとめて CRUD と呼びます．先程作ったテーブルで練習してみましょう．

まずは，データの追加から

```
sqlite> INSERT INTO student_records(student_id, klass, score, grade) VALUES(1, "
情報理工学実験", 90, "不可");
```

作成できたかを，データを取得してチェックしましょう．

```
sqlite> SELECT * FROM student_records;
  1|1|情報理工学実験|90|不可
```

90 点なのに不可はひどいですね．追加したデータを変更しましょう．

```
sqlite> UPDATE student_records SET grade = "良" WHERE student_id = 1;
sqlite> SELECT * FROM student_records;
  1|1|情報理工学実験|90|良
```

「WHERE〜」の部分は，更新対象のデータの指定条件を表します．

変更したばかりですが，次はデータを削除しましょう．

```
sqlite> DELETE FROM student_records WHERE id = 1;
sqlite> SELECT * FROM student_records;
```

もう少しデータ数を増やして遊んでみましょう．データが多いと，毎回 INSERT を入力するのは面倒なので，CSV ファイルからのインポートします．

まず，GoogleDocs から，以下の内容を含む二つのファイルの内容をコピーし，テキストファイルとして保存してください．ファイル名は以下のように設定してください．

student_records.csv : `https://drive.google.com/file/d/1DYEIN3KviWYP6AniwPgCCrH_VHV46Mhj/view?usp=sharing`

students.csv : `https://drive.google.com/file/d/1qvI-LiiYDLc6BcFeTQrc31WHDhbj_ulA/view?usp=sharing`

student_records.csv に含まれる内容は以下の通りです．

student_records.csv

```
1,1, 情報理工学実験,60, 可
2,2, 情報理工学実験,55, 不可
3,3, 情報理工学実験,20, 不可
4,4, 情報理工学実験,99, 秀
5,5, 情報理工学実験,40, 不可
6,6, 情報理工学実験,20, 不可
7,7, 情報理工学実験,30, 不可
8,8, 情報理工学実験,28, 不可
9,9, 情報理工学実験,1, 不可
10,1, 脳科学入門 IV,80, 良
11,2, 脳科学入門 IV,90, 秀
12,3, 脳科学入門 IV,10, 不可
13,4, 脳科学入門 IV,60, 可
```

students.csv には以下の内容が含まれています．

students.csv

```
1,John Smith
2, 山田太郎
3,Hans Schmidt
4, イワン・イワノビッチ・イワノフ
5, 佐藤一郎
6,John Doe
7, 後藤さん
8, 金さん
9, 熊さん
10, 権兵衛さん
666, でみあん
```

次に sqlite にデータをインポートします．

```
sqlite> .separator ,
sqlite> .import student_records.csv student_records
sqlite> .import students.csv students
sqlite> SELECT * FROM student_records;
```

separtor，import が sqlite の命令文であることに注意してください．

データが取得出来たか見てみましょう．以下の「–」で始まる行はコメントを表しています．

```
sqlite> -- 単位を落とした人を取得
sqlite> SELECT student_id, klass FROM student_records WHERE grade = "不可";
  2,情報理工学実験
  3,情報理工学実験
  5,情報理工学実験
  6,情報理工学実験
  7,情報理工学実験
  8,情報理工学実験
  9,情報理工学実験
  3,脳科学入門 IV
```

その他操作

SQL には他にも様々な機能があります．すべてを紹介するスペース・時間は到底ありませんが，特に有名な機能をいくつかご紹介します．

RDBMS では複数の表（テーブル）の集まりとしてデータを表しますので，時に，複数のテーブルを連結させたくなります．先のデータの例で言うと，成績表を見る時に，学生の名前も一緒のほうがわかりやすいですよね？連結を行う命令が「JOIN」です．連結の仕方により，INNER JOIN，LEFT JOIN，RIGHT JOIN，FULL OUTER JOIN などがあります（SQLite では RIGHT JOIN と FULL OUTER JOIN は実装されていません）．

使い方は，下記の例を参考にしてください．

```
sqlite> SELECT students.student_id,name,klass,grade FROM student_records INNER
JOIN students ON student_records.student_id = students.student_id;
  1,John Smith,情報理工学実験,可
  2,山田太郎,情報理工学実験,不可
  3,Hans Schmidt,情報理工学実験,不可
  4,イワン・イワノビッチ・イワノフ,情報理工学実験,秀
  5,佐藤一郎,情報理工学実験,不可
  6,John Doe,情報理工学実験,不可
  7,後藤さん,情報理工学実験,不可
  8,金さん,情報理工学実験,不可
  9,熊さん,情報理工学実験,不可
  1,John Smith,脳科学入門 IV,良
  2,山田太郎,脳科学入門 IV,秀
  3,Hans Schmidt,脳科学入門 IV,不可
  4,イワン・イワノビッチ・イワノフ,脳科学入門 IV,可
```

```
sqlite> SELECT students.student_id,name,klass,grade FROM students LEFT JOIN
student_records ON students.student_id = student_records.student_id;
  1,John Smith,情報理工学実験,可
  1,John Smith,脳科学入門 IV,良
  2,山田太郎,情報理工学実験,不可
  2,山田太郎,脳科学入門 IV,秀
  3,Hans Schmidt,情報理工学実験,不可
  3,Hans Schmidt,脳科学入門 IV,不可
  4,イワン・イワノビッチ・イワノフ,情報理工学実験,秀
  4,イワン・イワノビッチ・イワノフ,脳科学入門 IV,可
  5,佐藤一郎,情報理工学実験,不可
  6,John Doe,情報理工学実験,不可
  7,後藤さん,情報理工学実験,不可
  8,金さん,情報理工学実験,不可
  9,熊さん,情報理工学実験,不可
  10,権兵衛,,
  666,でみあん,,
```

使い方，INNER JOIN と LEFT JOIN の違いが確認出来ましたか？（データの順番は重要な違いではありません）

INNER JOIN では，連結するテーブルの双方に存在するデータを連結，取得します．LEFT JOIN では，左のテーブルに存在するデータは，右のテーブルに存在しなくても取得します．上の例で言いますと，students の「権兵衛」「でみあん」に関するデータは students_records にありませんが，LEFT JOIN では取得されています．

GROUP 命令では，名前の通りあるカラムの値でデータをグループ化し，各グループのデータを集計することが出来ます．こちらも，実際に見てもらった方が早いでしょう．各教科の最低，平均，最高点です．

```
sqlite> SELECT klass,MIN(score),AVG(score),MAX(score) FROM student_records
GROUP BY klass;
  情報理工学実験,1,39.2222222222222,99
  脳科学入門 IV,10,60.0,90
```

各学生の最低，平均，最高点です．

```
sqlite>select student_id,min(score),avg(score),max(score) from student_records
group by student_id;
  1,60,70.0,80
  2,55,72.5,90
  3,10,15.0,20
  4,60,79.5,99
  5,40,40.0,40
  6,20,20.0,20
  7,30,30.0,30
  8,28,28.0,28
  9,1,1.0,1
```

3.4 Ruby で DB

コネクタ

これまでの操作は SQLite のインタプリタを通して行ってきましたが，アプリケーションで使うには他のプログラミング言語から操作する必要があります．この他のプログラミング言語から RDBMS を操作するためのドライバはコネクタと呼ばれます．

幸いなことに，Ruby では本節の最初でインストールした sqlite3 の gem を用いることで，簡単に SQLite の操作を Ruby から行うことができます．

使う

まず，単純な例から見てみましょう

test_db.rb

```
# coding: utf-8
require 'sqlite3'
db = SQLite3::Database.new("sandbox.db")
db.execute("SELECT * FROM students") do |row|
  puts row.join(",")
end
db.close
```

「execute」メソッドで SQL 文を実行し，その結果が「row」という配列に格納されることになります．

パラメータを動的に指定出来るようにしてみましょう．

test_db2.rb

```
# coding: utf-8
require 'sqlite3'
db = SQLite3::Database.new("sandbox.db")
grade = gets.chomp
db.execute("SELECT student_id, klass FROM student_records WHERE grade = ?",
[grade]) do |row|
  puts row.join(",")
end
db.close
```

条件文の所の「?」の部分は，プレースホルダ（placeholder）と呼ばれる書き方で,「?」の部分を後続のパラメータを表す配列の要素で置き換えます．

ところで，こんな回りくどいやり方ではなく，前回説明した Ruby の文字列の埋め込みではいけないのか？と思う人もいるかもしれません．つまり，なぜ，下記のような書き方ではいけないのでしょうか？これが良くない書き方である理由は，基本課題 3-2 で確認しましょう．

test_db3.rb

```
require 'sqlite3'
db = SQLite3::Database.new("sandbox.db")
grade = gets.chomp
db.execute("SELECT student_id, klass FROM student_records WHERE grade =
\"#{grade} \"") do |row|
  puts row.join(",")
end
db.close
```

3.5 ActiveRecord による O/R マッパ

O/R マッパとは？

前述の通り，RDBMS を操作するには SQL を使うのがデフォルトです．SQL は問い合わせ型言語で，オブジェクト指向言語の Ruby や Java とは考え方が大きく異なり，言語の間の橋渡しが必要です．それを行うのが O/R マッパ（Object/Relational Mapper）で，Ruby の O/R マッパの実装（の一つ）が ActiveRecord です．この実験の後半で使う Sinatra ではオプションで使うことが出来ます．

また，これは O/R マッパとは別の話ですが，AcitveRecord を使うことで，DB の抽象化が行えるというメリットもあります．

使ってみる

ここでは，3.3 節で行った DB の基本動作を ActiveRecord を使って実現してみましょう．まずは，新しいテーブルの作成 (CREATE) から行います．

createNewTable.rb

```
# coding: utf-8
require 'active_record'
#DB と接続するためのメソッド
ActiveRecord::Base.establish_connection(
    "adapter" =>"sqlite3",         #adapter の指定 : mysql なども入っていれば設定可能
    "database" => "./ARLesson.db" #databese の指定 : ARLesson.db にアクセス可能とな
る
)

#新たなテーブル，student_records を作成
ActiveRecord::Migration.create_table :student_records do |t|
  t.integer :student_id  #カラムの名前と型を指定 ( t. 型名 :カラム名)
  t.text :klass
  t.integer :score
  t.text :grade
end
```

上記のコード中に「#カラムの名前と型を指定 (t. 型名 :カラム名)」というコメントが入っていますが，このコメントを残したまま実行するとこれ以降の作業でエラーになる場合があるようです．実際にコードを書く際にはこのコメントは削除して実行してください．ここで，ActiveRecord を使う上での注意点ですが，テーブル名は必ず複数形で作成してください.また，ActiveRecord にはいわゆる予約

語というテーブル名やカラム名に使用することができない文字列があります．個人の HP になりますが，予約語一覧が載っているサイトを紹介しておきます (URL: http://d.hatena.ne.jp/satake7/20080520)．

テーブルが出来たか一応見ておきましょう．sqlite のインタプリタで PRAGMA 命令を呼び出すことで，テーブルのカラムを確認出来ます．

```
$ sqlite3 ARLesson.db
sqlite> .mode line
sqlite> PRAGMA   TABLE_INFO (student_records);
         cid = 0
        name = id
        type = INTEGER
     notnull = 1
  dflt_value =
          pk = 1

         cid = 1
        name = student_id
        type = integer
     notnull = 0
  dflt_value =
          pk = 0

         cid = 2
        name = klass
        type = text
     notnull = 0
  dflt_value =
          pk = 0

         cid = 3
        name = score
        type = integer
     notnull = 0
  dflt_value =
          pk = 0

         cid = 4
        name = grade
        type = text
     notnull = 0
  dflt_value =
          pk = 0
```

テーブルの内容を確認してみると，最初のカラムに設定した覚えのない「id」というカラムがありますね．これは ActiveRecord を使って CREATE を行うと自動的に「id」という主キーを設定するためです．主キーとは，テーブルの中でレコードを一意に特定することのできるカラムを指します（例えば，大学で個人を特定するためには学生番号があれば，同姓同名の人物がいたとしても特定することができますよね）．そのため DB では主キーの存在を明示的に示す必要があるのです．AciveRecord は気を利かせて自動的に主キーを付与してくれるのです．便利ですね．

また，主キーの条件としては，

- 必ずデータが存在する　（NOT NULL）

- 他のデータと重複しない (UNIQUE)

ことが主にあります.

次に，作成したテーブルにデータを追加 (INSERT) し，それを確認してみましょう.

insertData.rb

```
# coding: utf-8
require 'active_record'
ActiveRecord::Base.establish_connection(
    "adapter" =>"sqlite3",
    "database" => "./ARLesson.db"
)

#ruby のクラスと DB 内のテーブルを結びつける (テーブル名の単数形をクラス名とする)
class Student_record < ActiveRecord::Base
end

#データを追加
Student_record.create(:student_id=> 1, :klass=>"情報理工学実験",
  :score=>90, :grade=>"不可")
Student_record.create(:student_id=> 2, :klass=>"情報理工学実験",
  :score=>70, :grade=>"良")
Student_record.create(:student_id=> 3, :klass=>"情報理工学実験",
  :score=>100, :grade=>"良")

#データの表示方法
puts '---全データを表示---'
p Student_record.all

puts '---:score の値が一番大きいデータを表示---'
p Student_record.order(score:"DESC").first

puts '---特定のデータを 1 つだけ表示---'
# 複数条件をつけると，AND で連結されます
p Student_record.find_by(:klass=>'情報理工学実験', :student_id => 1)    #.find_by(:
カラム名=>データ)

puts '---特定のデータを全て表示---'
p Student_record.where(:grade=>'良')          #.where(:カラム名 => データ)
```

上記のように，ActiveRecord は ruby のクラス名によって，DB 内のテーブル名と自動的に結びつけます．その際の大まかな規則は,「ruby のクラス名の複数形となる，テーブルと結びつける」です．従って，DB 内でテーブルを作成する際にはテーブル名を必ず複数形の単語にしてください．テーブルを表示させてみると「id」カラムは追加していないにも関わらず自動で付与されてますね．これは，DB の AUTO INCREMENT が主キーに設定されているからです．これも便利ですね．また，コードの後半では様々な方法でデータを抽出して表示をしています．これ以外にも様々な表示方法や組み合わせがたくさんあるので必要になったらインターネットなどで調べてください.

次はデータの変更 (UPDATE) です.

updateData.rb

```
# coding: utf-8

require 'active_record'

ActiveRecord::Base.establish_connection(
    "adapter" =>"sqlite3",
    "database" => "./ARLesson.db"
)

class Student_record < ActiveRecord::Base
end

#student_id が 1 のデータを取り出し
student_1 = Student_record.find_by(:student_id=>1)

#.update(:カラム名 => 更新後のデータ) で更新
student_1.update(:grade=>'秀')
```

ここでは，student_records の 1 つのデータを変数に保存しています．このように抽出したデータを変数に保存し，その処理を行うことも可能です．

最後に，データの削除 (DELETE) です．

deleteData.rb

```
# coding: utf-8

require 'active_record'

ActiveRecord::Base.establish_connection(
    "adapter" =>"sqlite3",
    "database" => "./ARLesson.db"
)

class Student_record < ActiveRecord::Base
end

#student_id が 3 のデータを消去
p Student_record.find_by(:student_id=>3).destroy

p Student_record.all
```

3.6 課題

下の基本課題全てと，応用課題を少なくとも一つ解きなさい．提出期限および提出先は Moodle 上で案内する．

各課題には，ソースコード（必要にして十分なコメントを付けて），実行結果例，その他説明を記入しなさい．

基本課題 3-1 NoSQL

今回の実験では RDBMS を勉強しましたが，RDBMS 以外にも DBMS は色々あり，それらは NoSQL と総称されます．NoSQL の例としては，Key-Value 型の Redis や KyotoCabinet，グラフデータベースの Neo4j，ドキュメント型の MongoDB などがあります．

NoSQL のいずれかについて調べ（上述の例以外も可），1〜3 ページ程度にまとめてください．

- RDBMS と比較した長所，短所
- 想定している用途
- 実際の使用例

基本課題 3-2 セキュリティ

データベースには各種データを保存しますので，そこに対する攻撃は重大な脅威となりえます．SQL を利用した攻撃方法の一つに，SQL インジェクション（SQL injection）があります．

SQL インジェクションについて調べ，

- 攻撃方法
- 下の weak_sqlite.rb への攻撃例
- 対策

をまとめてください．

weak_sqlite.rb

```
# coding: utf-8
require ‘sqlite3’
db = SQLite3::Database.new("sandbox.db")
id,new_name = gets.chomp.split(“,”)
str = "UPDATE students SET name = \"#{new_name}\" WHERE student_id = #{id}"
db.execute_batch(str)
db.close
```

基本課題 3-3 データベース作成

以下のデータベースを作り，テーブルに適当なデータを登録してください．レポートには作成，登録に使った SQL 文も記載しなさい．

labs テーブル	
id(integer PRIMARY KEY)	lab_name(text)

lab_members テーブル			
id(integer PRIMARY KEY)	lab_id(integer)	member_name(text)	position(text)

なお、簡単のためそれぞれのデータについては以下の csv ファイルを利用しても良い。

- labs.csv: https://drive.google.com/file/d/1zvfOBUOZzKGsHCqT4QUOG48tAxvWEEoQ/view?usp=sharing
- lab_members.csv: https://drive.google.com/file/d/1buNU7fgrqVABQy-CGHwFB6Q6tRz-0to3/view?usp=sharing

基本課題 3-4 データベース操作

基本課題 3-3 で作成した DB を使って，標準入力から DB のデータを閲覧出来るようにしてください．具体的には，以下の機能を実装してください．

- 研究室名 (lab_name) を入力すると，メンバー名 (member_name) 一覧が表示できる機能「**lab_search.rb**」
- メンバー名を入力すると，所属する研究室の他のメンバーの一覧が表示される機能「**member_search.rb**」

入力例：

```
$ ruby lab_search.rb
hci-lab # 入力
Tetsuo Ono
Daisuke Sakamoto
Masahiro Kitagawa

$ ruby member_search.rb
Daisuke Sakamoto # 入力
Ana_Forth
Greg_Woo
Dannette_Audia
<中略；表示される順番を変更しても良い>
Masahiro Kitagawa
Sho Mitarai
Tetsuo Ono
```

#補足
「その他の操作」で説明されていた 2 つのテーブルの連結ですが，異なるカラム名でも連結可能です．例えば，基本課題 3-3 のテーブルでは「labs.id」と「lab_members.lab_id」を使って連結することが可能です．

応用課題：諸注意

CGI ファイルを用いて DB のデータを Web ブラウザ上に表示する場合に日本語の文字が文字化けしてしまう場合には，DB のデータを英語に切り替えてください．

応用課題 3-1 データベース操作（Web 版）

基本課題 3-4 の Web 版です．Web ブラウザから DB のデータを閲覧できるようなプログラム「**labSearch_Server.rb**」を実装してください．具体的には，以下の機能を実装してください．

- 研究室名 (lab_name) をリクエストすると，メンバー名 (member_name) 一覧が取得出来る機能
- メンバー名をリクエストすると，所属する研究室の他のメンバーの一覧が表示される機能

リクエスト例：「127.0.0.1:2000/lab_search?lab_name=hci-lab」,
「127.0.0.1:2000/member_search?member_name=Daisuke Sakamoto」

応用課題 3-2 データベースを用いた掲示板

応用課題 2-2 の DB 版です．次の要件を満たす掲示板サーバ「**bulletinBoard_Server_withDB.rb**」を作りなさい．

1. サーバはクライアントから受け取った「ユーザー名」と「メッセージ」をデータベースに保存
2. クライアントはサーバに対し，書き込み「write」，書き込み一覧取得「index」，書き込み取得「read」のリクエストが出来る
3. 「write」は書き込むメッセージをパラメータとして受け取り，サーバはメッセージを保存し，ユニークな id を発行する
4. 「index」は保存されているユーザ名とメッセージの id 一覧を返す
5. 「read」は読み込む id をパラメータとして受け取り，その保存されている「ユーザ名」と「メッセージ」を返す

（要確認）実行例を下に示します．

```
http::://127.0.0.1:2000/write?user=yamada&msg=nullpo
http::://127.0.0.1:2000/write?user=yamada&msg=hoge
http::://127.0.0.1:2000/write?user=tanaka&msg=ga
http::://127.0.0.1:2000/index
yamada, 1
yamada, 2
tanaka, 3
http::://127.0.0.1:2000/read?id=1
yamada,nullppo
```

応用課題 3-3 データベースを用いたログイン機能

応用課題 2-3 の DB 版です．ただし，今度はユーザ機能を追加します．次の要件を満たすログイン機能を持つサーバを作りなさい．

- 事前にパスワードとユーザを管理するデータベースを作り，適当なデータを入れておく
- 「login」リクエスト「user」と「password」というパラメータを受け取り，データベースの内容と比較する
- 「password」で渡されたパスワードがデータベースに保存されている場合と，保存されていない場合で，区別出来るようにレスポンスを返す

入力のフォームには，以下の HTML を用い，Webrick もしくは Socket を用い，パラメータを受け取る login リクエストのプログラムを作成しなさい．

```
<!DOCTYPE html>
<html lang="ja">
<head>
<meta charset="UTF-8">
</head>
<body>
<form action="login" method="POST" >
<input type="text"  name="user" > <br>
<input type="text"  name="password" > <br>
<input type="submit" >
</form>
</body>
</html>
```

4 フレームワークを使った Web サーバプログラミング（5 日目，6 日目）

4.1 今回使う Ruby 構文

真偽値

いささか今更の話ですが，Ruby には真偽値 true と false があります．

```
puts “always to be called” if true
puts “never to be called” if false
```

Ruby では，**false と nil 以外は**真とみなします．C では 0 以外が真，Java では true だけが真で false だけが偽なのと混同しないように注意してください．

ActiveRecord の補足

ActiveRecotd で取得したデータは，各属性に簡単にアクセスすることが出来ます（下の例は実行しなくても構いません）．

```
> student = Student.find(2)
> puts student.name
> student.name = “Hoge hoge”
> student.save!
```

注意点として，where 文などではデータが配列（正確には配列みたいなもの）を返しますので，一個毎に取り出してアクセスする必要があります．

```
> student_list = Student.all
> student_list.each{ |s| puts s.name }
> puts student_list[0].name
```

4.2 セッションとクッキー

HTTP や TCP は状態を持たないプロトコルです．しかし，Amazon や Google などのサイトを利用する時，一回一回ユーザ名とパスワードを入力したりはしないと思います．このようなユーザの状態を管理するために使われているのが，セッションとクッキーです．

4.3 Web フレームワーク：Sinatra

Web フレームワークとは

Web フレームワーク（Web Application Framework: WAF）は，セキュリティやルーティング（クライアントからのリクエストを振り分ける機能），ログインなどの汎用的に必要な機能をまとめ，より簡単，安全，高速にプログラムを行うことを目的としたソフトウェア群です．代表的な WAF としては，PHP の CakePHP や Laravel，Ruby の Rails や Sinatra などがあります．

Sinatra とは

Ruby で作成された Web アプリケーションフレームワークである．（引用：Wikipedia）

ディレクトリ構成

Sinatra では，1 つのアプリケーションの中に様々なファイルやディレクトリが存在します．そのため，実習書においてどこにファイルを作成すれば良いか，どのファイルを編集すれば良いかについて，sinatra_app ディレクトリを root とした絶対パスで表記します．

4.4 Sinatra 準備〜入門

bundler 初期化

まずは，Sinatra アプリケーションのための sinatra_app ディレクトリを作成し，bundler の初期化を行います．

```
$ mkdir sinatra_app
$ cd sinatra_app
$ bundle init
```

sinatra_app に Gemfile が作成されたら，成功です．

bundler を使った Gem インストール

bundler を使って，Gem パッケージをインストールしましょう．準備として，先ほど作成した Gemfile を編集します．

Gemfile

```
# frozen_string_literal: true
source "https://rubygems.org"
#ーーーーーーーーーー追加部分ーーーーーーーーーー
gem 'sinatra',"1.4.0"
gem 'sinatra-contrib'
gem 'activerecord',"4.2.7"
gem 'sqlite3',"1.3.13"
#ーーーーーーーーーーーーーーーーーーーーーーーー
# gem "rails"
```

Gemfile に追加されたパッケージをインストールすることができるようになりました．それでは，以下のコマンドを入力してインストールしてみましょう．

```
$ bundle install --path=vendor/bundle
```

Web サーバを立ち上げる

後はプログラムを書くだけで，前章までと同じように Web サーバを立ち上げることができます．以下のコードを書いてみましょう（app.rb）．

app.rb

```
# coding: utf-8

require 'sinatra'
require 'sinatra/reloader'
require 'active_record'

get '/' do
  "Hello, World!"
end
```

それでは，Web サーバを立ち上げてみましょう．

```
$ bundle exec ruby app.rb
```

ブラウザで「127.0.0.1:4567」を開くとサーバが立ち上がっていることが確認することができます．今までと比べると簡単にサーバを立ち上げることができることが実感できますね．また，sinatra/reloader をインクルードしているので，コードを編集した際にサーバを立ち上げ直さなくても更新するだけで変更することが可能です．それでは，”Hello, World!”の部分を改変して試してみましょう．

ERB

次は ERB を使ってみましょう．ERB はテキストに Ruby のコードを埋め込むライブラリで，erb ファイルはテンプレートと一般的に呼ばれる種類です（テンプレートとしては YAML などがあります）．

これだけの説明だと，**ナンノコッチャ**ですが，さっそく使ってみましょう．まずは，views ディレクトリを作成します．

```
$ mkdir views
```

その後，views ディレクトリの中に index.erb ファイルを作成しましょう．index.erb と app.rb を次のように書き換えます．

app.rb の変更

```
# coding: utf-8

require 'sinatra'
require 'sinatra/reloader'
require 'active_record'

#ーーーーーーーーーーー変更部分ーーーーーーーーーーー
get '/' do
  @title = 'Hello, world!'
  @world = 'world!'
  erb :index
end
#ーーーーーーーーーーーーーーーーーーーーーーーーーー
```

以下は /views/index.erb の内容．

/views/index.erb

```
<html>
  <head>
    <title> <%= @title%> </title>
  </head>
  <body>
    <h1>Hello, <%=@world%> </h1>
  </body>
</html>
```

まず，app.rb では “erb :index” から，index.erb を呼び出しています．index.erb は一見普通の html 文書のように見えますが，<%=@world%>の部分で app.rb の変数を表示しています．ERB では，<%=ruby プログラム %>とすることで，プログラムの戻り値を表示することができます．また，<% ruby プログラム %>とすることでテキストにプログラムの記述を混ぜることができます．

一度 Web サーバを落としてしまった場合には、先ほどと同様に以下のコマンドで起動します。

```
$ bundle exec ruby app.rb
```

4.5 データベース (DB) と Sinatra の連携 –ログイン機能実装–

DB との連携

3 節で習った DB と Web サーバを連携してみましょう．いよいよ Web アプリケーションっぽくなってきましたね．DB は 3 節で使った student_records と students を使ってみましょう．2 つのテーブルを「**seiseki.db**」に保存してください．

（再掲）student_records.csv：`https://drive.google.com/file/d/1DYEIN3KviWYP6AniwPgCCrH_VHV46Mhj/view?usp=sharing`

（再掲）students.csv： `https://drive.google.com/file/d/1qvI-LiiYDLc6BcFeTQrc31WHDhbj_ulA/view?usp=sharing`

いまさらですが，以下のファイルを作成すると簡単にインポートすることができます．

import.sql

```
CREATE TABLE student_records(
        id integer PRIMARY KEY,
        student_id integer,
        klass text,
        score integer,
        grade text);

CREATE TABLE students(
        student_id integer PRIMARY KEY,
        name text);

.separator ,

.import student_records.csv student_records
.import students.csv students
```

作成した import.sql を以下のコマンドで読み込みます.

```
$sqlite3 seiseki.db
sqlite> .read import.sql
sqlite> SELECT * FROM student_records;
1,1, 情報理工学実験,60, 可
2,2, 情報理工学実験,55, 不可
3,3, 情報理工学実験,20, 不可
4,4, 情報理工学実験,99, 秀
5,5, 情報理工学実験,40, 不可
6,6, 情報理工学実験,20, 不可
7,7, 情報理工学実験,30, 不可
8,8, 情報理工学実験,28, 不可
9,9, 情報理工学実験,1, 不可
10,1, 脳科学入門 IV,80, 良
11,2, 脳科学入門 IV,90, 秀
12,3, 脳科学入門 IV,10, 不可
13,4, 脳科学入門 IV,60, 可
```

以上で準備完了です.

まずは，student_records の内容を公開するサーバを作ってみましょう．app.rb を次のように編集します.

app.rb に再度追加

```
# coding: utf-8

require 'sinatra'
require 'sinatra/reloader'
require 'active_record'

#ーーーーーーーーーー追加部分ーーーーーーーーーー
ActiveRecord::Base.establish_connection(
  adapter: 'sqlite3',
  database: 'seiseki.db'
)

class StudentRecord < ActiveRecord::Base
end
#ーーーーーーーーーーーーーーーーーーーーーーーーー

get '/' do
  @title = 'Hello, world!'
  @world = 'world!'
  erb :index
end

#ーーーーーーーーーー追加部分ーーーーーーーーーー
get '/seiseki' do
  @title = 'Seiseki View'
  @students = StudentRecord.all
  erb :seiseki
end
#ーーーーーーーーーーーーーーーーーーーーーーーーー
```

その後，新しい ERB ファイル seiseki.erb を次のように作成しましょう．

seiseki.erb

```
<!DOCTYPE html>
<html lang="ja">
  <head>
    <meta charset="utf-8">
    <title><%= @title %></title>
  </head>
  <body>
    <h1> <%= @title %> </h1>
    <table border>
      <tr>
        <th>id</th>
        <th>student_id</th>
        <th>klass</th>
        <th>grade</th>
      </tr>
      <% @students.each do |stu| %>
      <tr>
        <td><%= stu.id %></td>
        <td><%= stu.student_id %></td>
        <td><%= stu.klass %></td>
        <td><%= stu.grade %></td>
      </tr>
      <% end %>
    </table>
  </body>
</html>
```

app.rb では，ActiveRecord を呼び出し，student_records の全データを保存して，その後 seiseki.erb ファイルに渡しています．seiseki.erb では，受け取った student_records のデータを表にして表示しています．「127.0.0.1:4567/seiseki」にアクセスして確認してみましょう．

これで「DB と Web サーバを連携できて万歳!」と言いたいところなのですが，このままだと誰でも成績を閲覧できてしまうので大変ですね．なので，次はログイン機能を作りましょう．

ログイン機能実装

ログイン機能の情報として，students テーブルを利用します．入力フォームに id と名前を入力し，両方がマッチしていたら，ログインできる，という仕組みです．まず，入力フォームを作成するために，index.erb を次のように編集してください．

index.erb の編集

```
<!DOCTYPE html>
<html lang="ja">
  <head>
    <meta charset="utf-8">
    <title><%= @title %></title>
  </head>
  <body>
    <h1>Hello, <%=@world%> </h1>
    <table>
      <tr>
        <td>student_id</td>
        <td>name</td>
      </tr>

      <form method="post" action="judge">
        <tr>
          <td><input type="text" name="student_id"></td>
          <td><input type="text" name="name"></td>
          <td><input type="submit" value="ログイン"></td>
        </tr>
      </form>
    </table>
  </body>
</html>
```

上のファイルでは，POST リクエストにより student_id と name のデータを「127.0.0.1:4567/judge」へと送ることができます．次に，app.rb を以下のように編集してください．

app.rb の編集

```
# coding: utf-8

require 'sinatra'
require 'sinatra/reloader'
require 'active_record'

ActiveRecord::Base.establish_connection(
  adapter: 'sqlite3',
  database: 'seiseki.db'
)

class StudentRecord < ActiveRecord::Base
end

#ーーーーーーーーーー追加部分ーーーーーーーー
class Student < ActiveRecord::Base
end
#ーーーーーーーーーーーーーーーーーーーーーー

get '/' do
  @title = 'Login Form' #変更部分
  @world = 'World!'
  erb :index
end

#ーーーーーーーーーー追加部分ーーーーーーーー
post '/judge' do
  if !Student.where(:student_id => params[:student_id], :name => params[:name])
.empty?
    @title = 'View Seiseki'
    @student = StudentRecord.where(:student_id => params[:student_id])
    @student_name = params[:name]
    erb :login
  else
    redirect '/'
  end
end
#ーーーーーーーーーーーーーーーーーーーーーー
```

このファイルでは，post リクエストで受け取ったデータを処理し，適切なページを表示しています．最後にログイン後の画面として，自分の ID の成績のみを表示します．以下の/views/login.erb を作成しましょう．

/views/login.erb

```
<!DOCTYPE html>
<html lang="ja">
  <head>
    <meta charset="utf-8">
    <title><%= @title %></title>
  </head>
  <body>
    <h1>Hello!, <%= @student_name %> </h1>
    <h2>Your Grade is blow.</h2>
    <table border>
      <tr>
        <th>id</th>
        <th>student_id</th>
        <th>klass</th>
        <th>grade</th>
      </tr>
      <% @student.each do |stu| %>
      <tr>
        <td><%= stu.id %></td>
          <td><%= stu.student_id %></td>
          <td><%= stu.klass %></td>
          <td><%= stu.grade %></td>
      </tr>
      <% end %>
    </table>
  </body>
</html>
```

しかし，このコードにはいくつか（実はいくつも）セキュリティ上の問題点があります．ID と名前だけだったら簡単に破られてしまうなど低レベルな問題もありますが，それより大事な問題は **POST リクエストを送る際にパラメータを傍受されてしまう可能性があることです**．つまり，今の状態ではどれだけ強固なパスワードを使ったとしても，通信を盗み見られてしまうことで簡単に不正アクセスされてしまうのです．この点について，後述する課題で対策を学びましょう．

4.6 課題

下の基本課題全てと，応用課題を少なくとも一つ解きなさい．また，それらに取り組む代わりに発展課題を解いても良い．各課題には，ソースコード（必要にして十分なコメントを付けて），実行結果例，その他説明を記入しなさい．提出期限および提出先は Moodle 上で案内する．

基本課題 4-1，応用課題 4-1 と応用課題 4-2 の画面設計については以下のモックアップを参考にしてください。ただし、画面設計は必ずしもこのモックアップにあわせる必要はありません。

`https://drive.google.com/drive/folders/16pPpK9c_0vc5YrIKEeVqQkjf-S_FzyzN?usp=sharing`

基本課題 4-1 Web フレームワークを利用したデータベース操作

応用課題 3-1 を Sinatra を用いて実装しなさい．課題内容は，Web ブラウザから DB のデータを閲覧できるようにすることです．具体的には，以下の機能を実装してください．

- 研究室名 (lab_name) をリクエストすると，メンバー名 (member_name) 一覧が取得出来る機能

- メンバー名をリクエストすると，所属する研究室の他のメンバーの一覧が表示される機能

リクエスト例：「127.0.0.1:2000/lab_search?lab_name=hci-lab」，
「127.0.0.1:2000/member_search?member_name=Daisuke Sakamoto」

課題で作成するプログラムは以下の 4 つのファイルです。これらをレポートとして提出してください。

- メイン：app.rb
- 検索画面テンプレート：index.erb
- 研究室名をリクエストするとメンバー名の一覧を表示するテンプレート：lab_search.erb
- メンバー名をリクエストすると，所属する研究室のメンバー名の一覧を表示するテンプレート：member_search.erb

基本課題 4-2 認証

利用者が限定されるような Web ページでは,「認証」（Authentication）という仕組みを利用して利用者を制限します．よく利用される認証方法である，Basic 認証・ダイジェスト認証について調べ，1〜3 ページ程度にまとめなさい．その際，これまでの HTML フォームを利用するログインとの違い，それぞれの長所短所などを明記しなさい．

基本課題 4-3 REST

REST という設計思想があり，Web サービスなどでは基本的にそれに沿って作成するのが，グッドマナーです．REST について調べ，1 ページから 3 ページ程度にまとめなさい．

応用課題 4-1 Web アプリケーションの作成①

Sinatra を使って，session を用いたログイン機能を持った掲示板を作成しなさい．(ただし，ダイジェスト認証は使わなくても良い)

#今回は session を用いた通信を行ってください．（cookie では改ざんの可能性）
session の特徴

- サーバー側に保存
- データ改竄の可能性は低い
- 取り出すには sessionID をサーバーに送信する必要あり

cookie の特徴

- クライアント側に値を保存
- データ改竄の可能性あり

#sinatra で session を使うには？
=>sinatra ではデフォルトでオフになっているため，sinatra で利用するためには，プログラム内で使うことが出来るように設定しなくてはなりません．session の使い方について調べてみてください．
例） ログイン画面でユーザ ID とパスワードを入力（実習書ではログイン機能は実装済）

=> 一致したらログイン後の掲示板ページを表示
一致しなかったら元のログインページを表示（「ログインできませんでした」と表示）

=> 掲示板ページでは，今までに入力されたメッセージをユーザ名と一緒に閲覧する
入力フォームも用意し，入力したメッセージを自分のユーザ名，現在時刻と一緒に追加可能

応用課題 4-2 Web アプリケーションの作成②

Sinatra を使ってお気に入りの Web サイトを DB に保存するアプリケーションを作成しなさい（Pocket や，はてなブックマークのイメージです）．その際に DB の構成（テーブル名，カラム名）も自分で考えなさい．また，最低でも，以下の条件を満たすこと．

- ログイン機能を持つ
- ログインすると，ページ名と URL の表を閲覧可能
- ログインしたページで，新たな情報をデータベースに登録できる
- ログインしたページから，お気に入りの Web サイトへアクセスできる（リンクが貼ってある）

課題で作成するプログラムは以下の 3 つのファイルです。これらをレポートとして提出してください。

- メイン：app.rb
- ログイン画面のテンプレート：index.erb
- お気に入りの Web サイト登録画面のテンプレート：weblist.erb

発展課題 Web フレームワークと JavaScript を利用した同期型アプリケーション

Sinatra を使って，お絵かきを共有 (お互いに編集可) できる Web アプリケーションを作りなさい（GoogleDocs のペイント版のようなイメージ）．使用したライブラリ，ソースコードの説明，実装した機能，作成したアプリケーションの画面のスクリーンショットを記載しなさい．#必要な知識

- HTML5 の Canvas, Websocket, JavaScript 等々？